华东师范大学计算机科学与技术学院实验报告

课程名称：数据挖掘	年级：2020级	实践作业成绩：
指导教师：兰曼	姓名：张宏伟	提交作业日期：2023/5/29
实践编号：2	学号：10205102417	实践作业编号：2

华东师范大学计算机科学与技术学院实验报告
一、 实验名称：手写数字识别
二、 实验目的
三、 实验内容
1. 实用工具
2. 图片数据集处理
2.1 数据集选取
2.2 数据填充
2.3 数据集shuffle处理
3. 识别算法
3.1 KNN
3.2 CNN
3.2.1 单层卷积
3.2.2 双层卷积
3.2.3 三层卷积
3.2.4 四层卷积
3.2.5 训练模型
3.3 LSTM
3.3.1 网络设计
3.3.2 模型训练
3.4 MLP
3.5 随机森林
四、 实验结果及分析
1. KNN
2. CNN
3. LSTM
4.MLP 和 随机森林
5. 对比
五、 问题讨论
六、 结论

一、 实验名称：手写数字识别

提供的手写数字图片数据集主要由一些手写数字的图片和相应的标签组成，图片一共有10 类，分别对应从0～9，共10 个阿拉伯数字，需要对其进行分类

二、 实验目的

掌握图片读取、转化为矩阵的方法，学习分类算法

三、 实验内容

1. 实用工具

- sklearn
- pytorch

2. 图片数据集处理

2.1 数据集选取

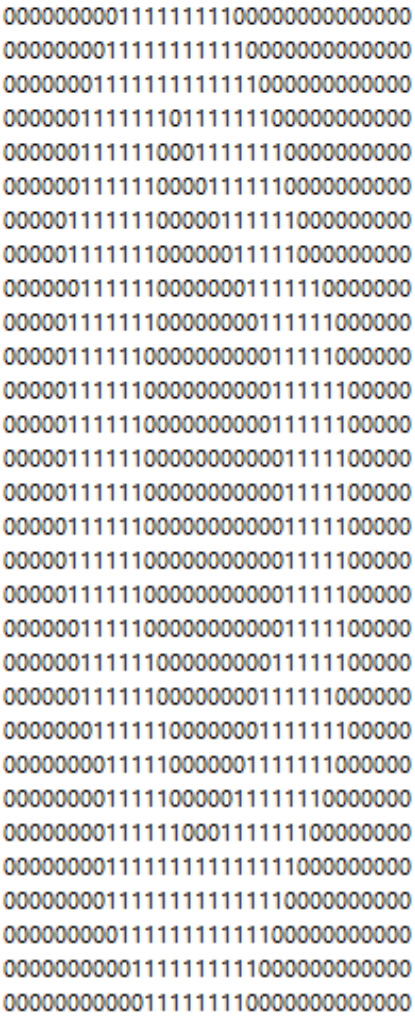
本次作业老师提供了1000+的手写数字图片训练集和400+的测试集

在实验过程中，为例进一步提高模型性能，引入了 **MNIST** 手写数字图片数据集 (http://yann.lecun.com/exdb/mnist/)，其中包含60000张图片的训练集和10000张图片的测试集。将两者进行了合并，最终数据集情况如下：

- 训练集：61498
- 测试集：10434

2.2 数据填充

由于MNIST的数据集为28*28像素的图片，而老师提供的数据集为32*32，在后续实验中，使用CNN和LSTM进行分类时，需要统一大小。观察老师提供的数据集，发现数字填充较满，裁剪掉二行二列可能会破坏特征信息，于是对MNIST数据集进行二行二列的0填充。



使用pytorch中pad函数对训练集和数据集tensor进行0填充

```
train_data=torch.nn.functional.pad(train_data,(2,2,2,2,0,0),"constant")
```

2.3 数据集shuffle处理

在使用CNN和LSTM进行测试时，发现仅仅使用老师所给的训练集时，测试效果并不理想（准确率在40%~50%）。观察数据集，发现训练集中的数字分布均为连续性的，即连续150个0接上连续150个1。推测是因为训练数据的连续分布，使得**模型错误地学习到了一定的序列特征**（特别是对于**LSTM模型**）。因此，要对训练集进行shuffle处理。

MNIST数据集通过torch.dataloader模块读取，数据和标签一一对应，可以直接shuffle。但是，老师提供的数据手动读取pytorch.tensor类型，无法直接转化为dataloader，不能直接shuffle。

这里使用下标的方式，标记数据tensor和标签tensor，对下标进行shuffle后，通过下标读取数据和对应的标签。达到了破坏原有的序列特征的目的。

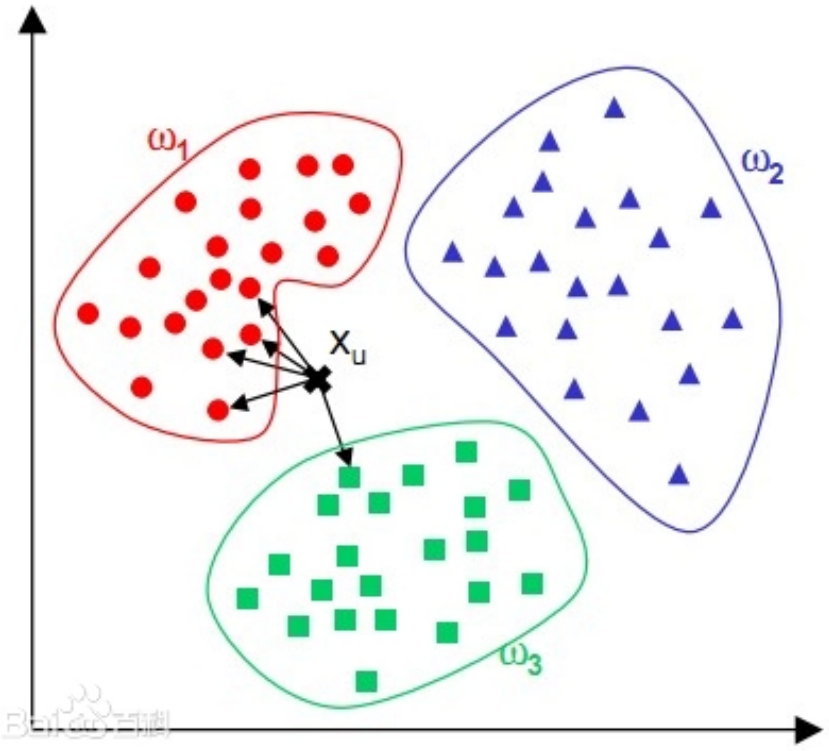
```
label_train = []
data_len_train = getTrainData.getDataLen()
data_train = torch.zeros(data_len_train, 1, 32, 32)
data_train, label_train = getTrainData.createTrainData()
tag = []

for i in range(data_len_train):
    tag.append(i)

random.shuffle(tag)
```

3. 识别算法

本次实验分别尝试了五种识别算法：**KNN**、**CNN**、**LSTM**、**MLP**、**随机森林**

3.1 KNN

KNN(K Nearest Neighbors) 算法，当预测一个新的值x的时候，根据它距离最近的K个点是什么类别来判断x属于哪个类别，KNN算法中只有一个超参数k，k值的确定对KNN算法的预测结果有着至关重要的影响。



实验中使用sklearn封装的KNN算法进行手写数字分类

- **将数据集转化为numpy矩阵**

```
def img2vect(filename):
    returnVect = zeros((1, 1024))  # 首先创建1×1024的Numpy数组
    fr = open(filename)
    for i in range(32):  # 循环读出文件的前32行，并将每行的前32个字符值存储在Numpy数组中
        lineStr = fr.readline()
        for j in range(32):
            returnVect[0, 32*i + j] = int(lineStr[j])
    return returnVect  # 最后返回数组
```

- **加载训练集**

```
def load_trainingData():
    hwLabels = []
    trainingFileList = listdir('../data/trainingDigits')  # 获取训练目录的内容
    m = len(trainingFileList)  # 获取训练样本个数
    trainingMat = zeros((m, 1024))  # 矩阵每行存储一个train图像
    for i in range(m):
        fileNameStr = trainingFileList[i]  # 从训练数据文件名解析分类数字
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        hwLabels.append(classNumStr)
        trainingMat[i, :] = img2vect('../data/trainingDigits/%s' % fileNameStr)
    return trainingMat, hwLabels
```

- **加载测试集**

```
def load_testData():
    hwLabels = []
    trainingFileList = listdir('../data/testDigits')  # 获取测试目录的内容
    m = len(trainingFileList)  # 获取测试样本个数
    testMat = zeros((m, 1024))  # 矩阵每行存储一个test图像
    for i in range(m):
        fileNameStr = trainingFileList[i]  # 从测试数据文件名解析分类数字
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        hwLabels.append(classNumStr)
        testMat[i, :] = img2vect('../data/testDigits/%s' % fileNameStr)
    return testMat, hwLabels
```

- **加载待预测数据集**

```
def load_predictData():
    trainingFileList = listdir('../testfile_7/testfile_7')
    m = len(trainingFileList)
    testMat = zeros((m, 1024))
    for i in range(m):
        fileNameStr = trainingFileList[i]
        testMat[i, :] = img2vect('../testfile_7/testfile_7/%s' % fileNameStr)
    return testMat
```

- **定义分类器**

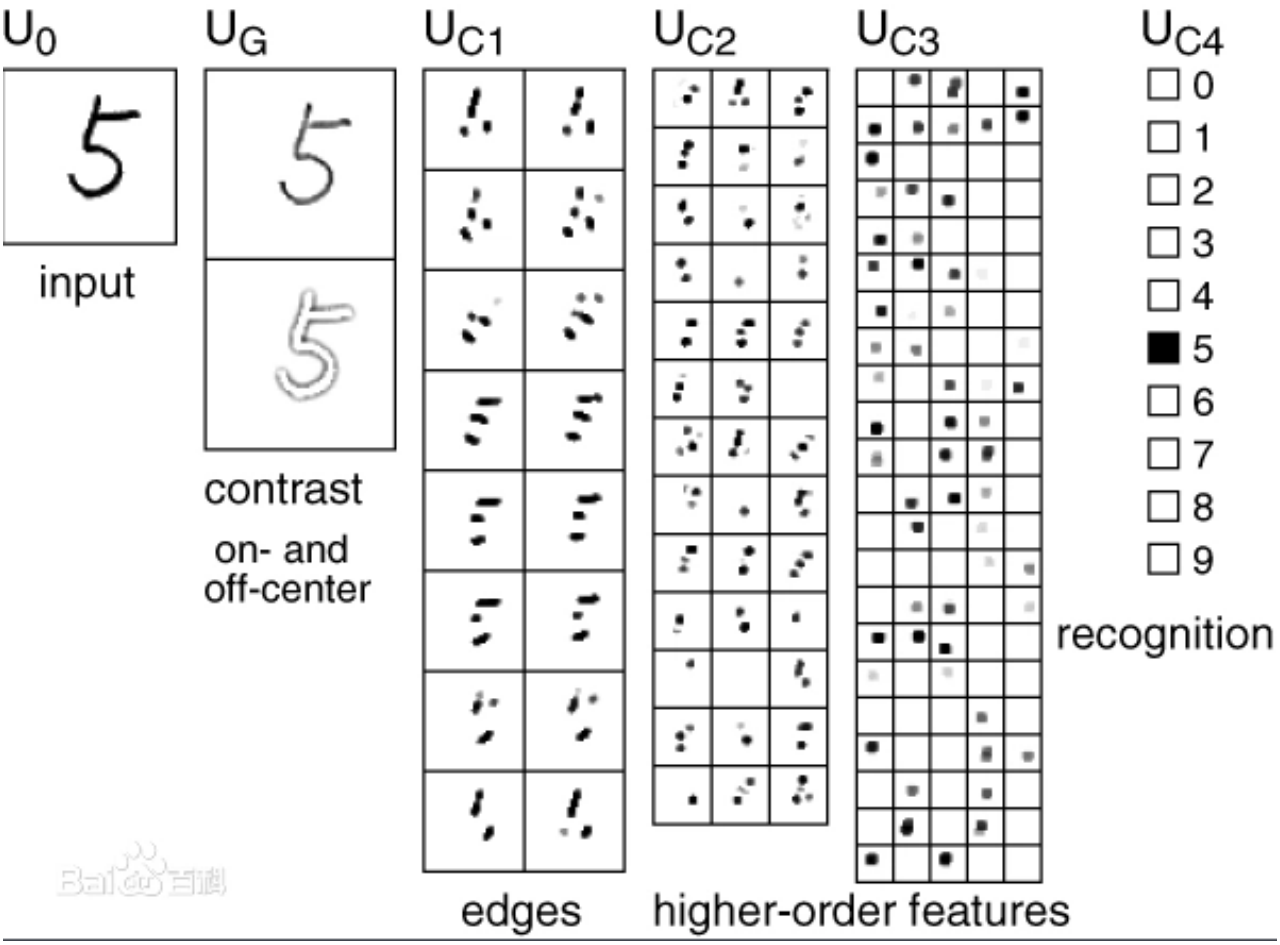
```
def predictor(k):
    trainingMat, hwLabels = load_trainingData()
    testMat, goldLabels = load_testData()
    predictMat = load_predictData()
    mTest = len(testMat)
    classifierResult = []
    clf = KNeighborsClassifier(algorithm='kd_tree', n_neighbors=k)
    clf.fit(trainingMat, hwLabels)
    classifierResult = clf.predict(predictMat)
    return classifierResult
```

3.2 CNN

卷积神经网络(CNN)是一类包含卷积计算且具有深度结构的前馈神经网络，通常包含以下结构：

- **卷积层**（Convolutional layer）
- **激活层**（Rectified Linear Units layer, ReLU layer）
- **池化层**（Pooling layer）
- **全连接层**（ Fully-Connected layer）

CNN通过卷积层抽取特征信息，再通过池化层下采样，保留特征信息相对位置，可以高效进行分类如图显示CNN进行分类的步骤可视化：



本次实验中，利用pytorch进行卷积神经网络的构建，分别设计了单层卷积、双层卷积、三层卷积和四层卷积来进行对比实验。

3.2.1 单层卷积

网络设计如下：

卷积+ReLu+池化 —> 全连接

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.out = nn.Linear(16 * 16 * 16, 10)

    def forward(self, x):
```

```
        x = self.conv1(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output
```

3.2.2 双层卷积

网络设计如下：

卷积+ReLu+池化 —> 卷积+ReLu+池化 —> 全连接

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=16,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.out = nn.Linear(32 * 8 * 8, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output
```

3.2.3 三层卷积

网络结构设计如下：

卷积+ReLu+池化 —> 卷积+ReLu —> 卷积+ReLu+池化 —> 全连接

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=16,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(
                in_channels=16,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.out = nn.Linear(16 * 8 * 8, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
```

```
        return output
```

3.2.4 四层卷积

网络结构设计如下：

卷积+ReLu+池化 —> 卷积+ReLu —> 卷积+ReLu+池化 —> 卷积+ReLu —> 全连接

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=16,
                out_channels=32,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(
                in_channels=32,
                out_channels=64,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(
                in_channels=64,
                out_channels=64,
```

```
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
        )
        self.out = nn.Linear(64 * 8 * 8, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output
```

3.2.5 训练模型

训练网络是选用**Adam**优化器，使其能更好的适用于稀疏梯度的情况。

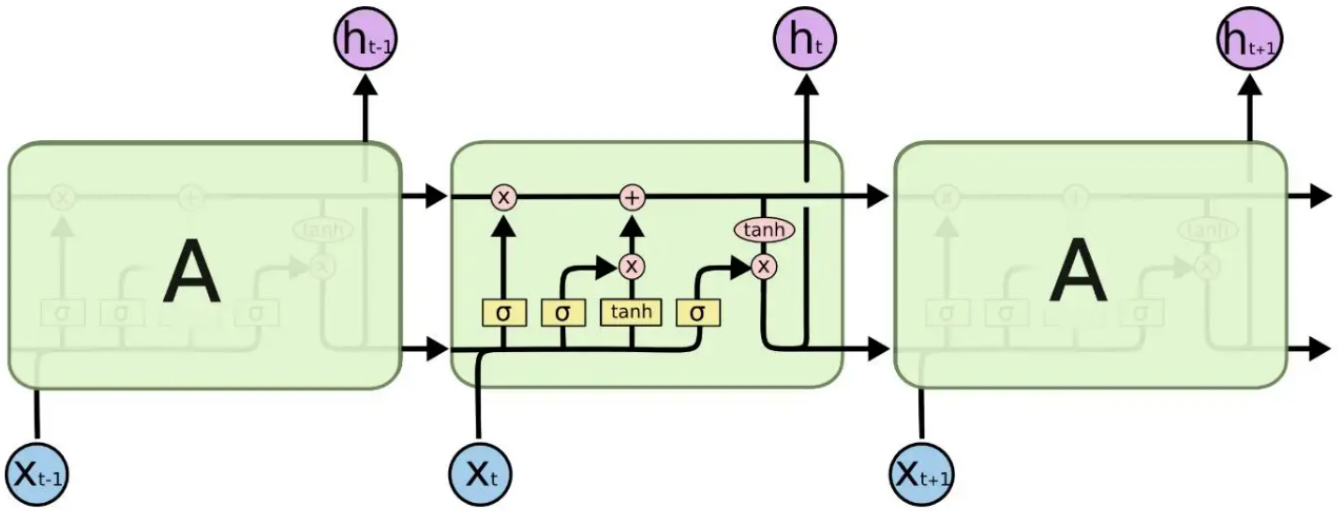
```
def train():
    label_train = []
    data_len_train = getTrainData.getDataLen()
    data_train = torch.zeros(data_len_train, 1, 32, 32)
    data_train, label_train = getTrainData.createTrainData()

    print(data_train.size())
    print(label_train.size())

    # 使用下标来shuffle
    tag = []
    for i in range(data_len_train):
        tag.append(i)
    random.shuffle(tag)

    label_test = []
    data_len_test = getTrainData.getDataLen()
    data_test = torch.zeros(data_len_test, 1, 32, 32)
    data_test, label_test = getTrainData.createTrainData()

    cnn = CNN()
    print(cnn)
    cnn = cnn.to(device)

    # 优化器选择Adam
```

```
    optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)

    # 损失函数
    loss_func = nn.CrossEntropyLoss()

    # 开始训练
    for epoch in range(EPOCH):
        for step, (b_x, b_y) in enumerate(train_loader):
            print(b_x.size())
            b_x = F.pad(b_x, (2, 2, 2, 2), "constant")
            b_y = b_y.to(device)
            output = cnn(b_x)

            # 先将数据放到cnn中计算output
            loss = loss_func(output, b_y)

            optimizer.zero_grad()  # 清除之前学到的梯度的参数
            loss.backward()  # 反向传播，计算梯度
            optimizer.step()

        test_output = cnn(data_test)
        pred_y = torch.max(test_output, 1)[1].data.cpu().numpy()
        accuracy = float((pred_y == label_test.data.cpu().numpy()).astype(int).sum()) /
float(label_test.size(0))

        print('Epoch: ', epoch+1, '| train loss: %.5f' % loss.data.cpu().numpy(), '|
test accuracy: %.5f' % accuracy)

    torch.save(cnn.state_dict(), 'cnn_new.pkl')  # 保存模型
```

3.3 LSTM

LSTM（Long Short-Term Memory）长短记忆神经网络，是一种时间循环神经网络。LSTM引入了门（gate）机制用于控制特征的流通和损失。LSTM的表现通常比时间递归神经网络及隐马尔科夫模型（HMM）更好，比如用在不分段连续手写识别上

LSTM单元的结构如下图所示：



The repeating module in an LSTM contains four interacting layers.

3.3.1 网络设计

使用torch.nn中封装的LSTM单元，使用两层LSTM，最后输入至全连接层，进行分类。

```
class rnn_classify(nn.Module):
    def __init__(self, in_feature=28, hidden_feature=100, num_class=10, num_layers=2):
        super(rnn_classify, self).__init__()
        self.rnn = nn.LSTM(in_feature, hidden_feature, num_layers)  # 使用两层LSTM
        self.classifier = nn.Linear(hidden_feature, num_class)

    def forward(self, x):
        x = x.squeeze()  # 去掉（batch, 1,28,28）中的1，变成（batch,  28,28）
        x = x.permute(2, 0, 1)  # 将最后一维放到第一维，变成（batch, 28,28）
        out, _ = self.rnn(x)  # 使用默认的隐藏状态，得到的out是（28,  batch,
hidden_feature）
        out = out[-1, :, :]
        out = self.classifier(out)
        return out
```

3.3.2 模型训练

对模型进行10个EPOCH的训练，输出每个EPOCH训练中的LOSS，以及在测试集上的准确率。

```
def train(net, train_data, valid_data, num_epochs, optimizer, criterion):
    if torch.cuda.is_available():
        net = net.cuda()

    prev_time = datetime.datetime.now()

    for epoch in range(num_epochs):
```

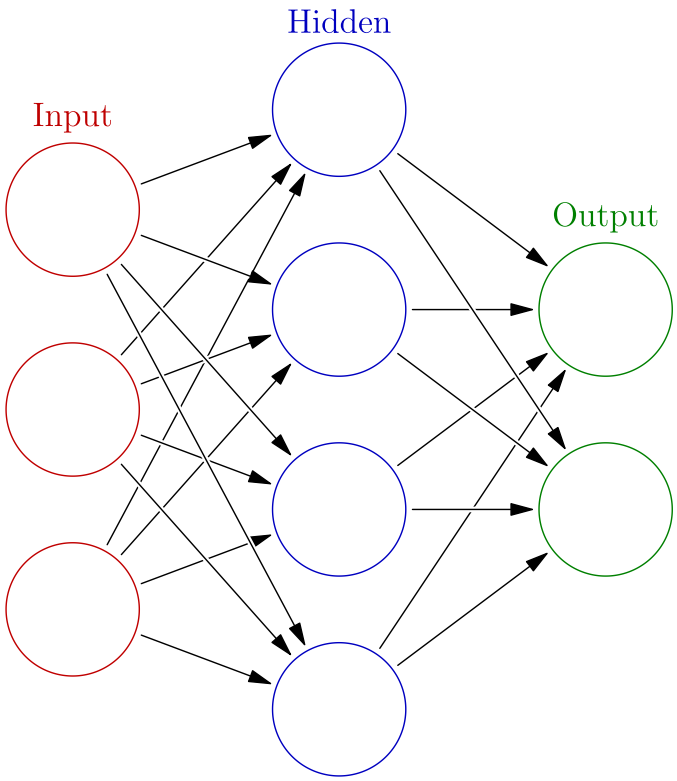
```
        train_loss = 0
        train_acc = 0
        net = net.train()

        for im, label in train_data:
            if torch.cuda.is_available():
                im = Variable(im.cuda())  # (bs, 3, h, w)
                label = Variable(label.cuda())  # (bs, h, w)
            else:
                im = Variable(im)
                label = Variable(label)

            # Forward
            output = net(im)
            loss = criterion(output, label)

            # Backward
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            train_acc += get_acc(output, label)

        cur_time = datetime.datetime.now()
        h, remainder = divmod((cur_time - prev_time).seconds, 3600)
        m, s = divmod(remainder, 60)
        time_str = "Time %02d:%02d:%02d" % (h, m, s)

        if valid_data is not None:
            valid_loss = 0
            valid_acc = 0
            net = net.eval()

            for im, label in valid_data:
                if torch.cuda.is_available():
                    im = Variable(im.cuda())
                    label = Variable(label.cuda())
                else:
                    im = Variable(im)
                    label = Variable(label)

                output = net(im)
                loss = criterion(output, label)

                valid_loss += loss.item()
                valid_acc += get_acc(output, label)

            epoch_str = (
```

```
                "Epoch %d. Train Loss: %f, Train Acc: %f, Valid Loss: %f, Valid Acc:
%f, " % (
                    epoch, train_loss / len(train_data), train_acc / len(train_data),
                    valid_loss / len(valid_data), valid_acc / len(valid_data)
                )
            )
        else:
            epoch_str = (
                "Epoch %d. Train Loss: %f, Train Acc: %f, " % (
                    epoch, train_loss / len(train_data), train_acc / len(train_data)
                )
            )

        prev_time = cur_time
        print(epoch_str + time_str)
```

3.4 MLP

多层感知机（Multilayer Perceptron，简称 MLP）是一种常用的前馈神经网络模型，用于解决分类和回归问题。它由多个神经网络层组成，每个层都包含多个神经元（也称为节点）。MLP 的每个神经元与前一层的所有神经元相连接，每个连接都有一个权重。神经元通过激活函数将加权输入进行非线性转换，然后将输出传递给下一层。

MLP 的基本结构如下所示：




```
import os
import numpy as np
```

```
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

# 1. 数据加载和预处理
def load_data(directory):
    images = []
    labels = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            label, _ = filename.split("_")
            with open(os.path.join(directory, filename), "r") as file:
                image_data = file.read().replace('\n', '')
            image = np.array([int(pixel) for pixel in image_data])
            images.append(image)
            labels.append(int(label))
    return np.array(images), np.array(labels)

train_images, train_labels = load_data("trainDigits")  # 根据实际路径修改

# 2. 拆分训练集和验证集
X_train, X_val, y_train, y_val = train_test_split(train_images, train_labels,
test_size=0.2, random_state=42)

# 3. 训练模型
model = MLPClassifier(hidden_layer_sizes=(128, 64), max_iter=500)  # 使用多层感知机作为示例
模型，可以根据需要选择其他算法
model.fit(X_train, y_train)

# 4. 在验证集上评估模型性能
accuracy = model.score(X_val, y_val)
print("Validation Accuracy:", accuracy)

# 5. 加载和预测测试数据
def load_test_data(directory):
    test_images = []
    test_filenames = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            with open(os.path.join(directory, filename), "r") as file:
                image_data = file.read().replace('\n', '')
            test_image = np.array([int(pixel) for pixel in image_data])
            test_images.append(test_image)
            test_filenames.append(filename.split(".")[0])
    return np.array(test_images), test_filenames

test_images, test_filenames = load_test_data("testfile_7")  # 根据实际路径修改

# 6. 进行预测并保存结果
```

```
predictions = model.predict(test_images)
sorted_indices = np.argsort(np.array(test_filenames, dtype=int))
sorted_filenames = np.array(test_filenames)[sorted_indices]
sorted_predictions = predictions[sorted_indices]

output_filename = "10205102417-predictions.txt"  # 根据实际学号修改
with open(output_filename, "w") as file:
    for filename, prediction in zip(sorted_filenames, sorted_predictions):
        file.write(str(filename) + " " + str(prediction) + "\n")

print("Predictions saved to:", output_filename)
```

3.5 随机森林

随机森林（Random Forest）是一种集成学习方法，通过构建多个决策树并综合它们的预测结果来进行分类或回归任务。它是由多个决策树组成的集成模型，每个决策树都是通过对数据集进行有放回抽样（bootstrap sampling）得到的不同训练集训练而成。

```
import os
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# 1. 数据加载和预处理
def load_data(directory):
    images = []
    labels = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            label, _ = filename.split("_")
            with open(os.path.join(directory, filename), "r") as file:
                image_data = file.readlines()
            image = np.array([[int(pixel) for pixel in line.strip()] for line in
image_data])
            images.append(image.flatten())
            labels.append(int(label))
    return np.array(images), np.array(labels)

def load_test_data(directory):
    test_images = []
    test_filenames = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            with open(os.path.join(directory, filename), "r") as file:
                image_data = file.readlines()
```

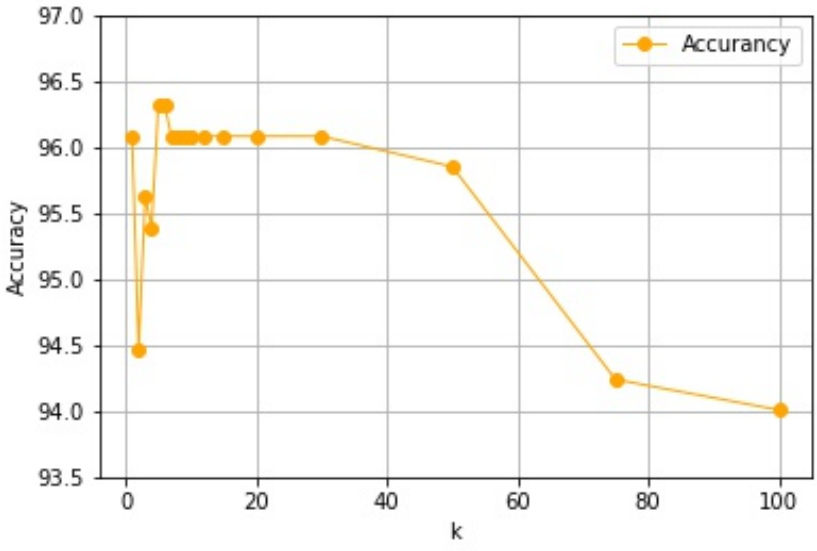
```
            test_image = np.array([[int(pixel) for pixel in line.strip()] for line in
image_data])
            test_images.append(test_image.flatten())
            test_filenames.append(filename.split(".")[0])
    return np.array(test_images), test_filenames

# 2. 加载和预处理数据集
train_images, train_labels = load_data("trainDigits")  # 根据实际路径修改
X_train, X_val, y_train, y_val = train_test_split(train_images, train_labels,
test_size=0.2, random_state=42)

# 3. 创建和训练随机森林模型
model = RandomForestClassifier(n_estimators=100, random_state=42)  # 根据需要设置随机森林的
参数
model.fit(X_train, y_train)

# 4. 在验证集上评估模型性能
accuracy = model.score(X_val, y_val)
print("Validation Accuracy:", accuracy)

# 5. 加载和预测测试数据
test_images, test_filenames = load_test_data("testfile_7")  # 根据实际路径修改

# 6. 进行预测并保存结果
predictions = model.predict(test_images)
sorted_indices = np.argsort(np.array(test_filenames, dtype=int))
sorted_filenames = np.array(test_filenames)[sorted_indices]
sorted_predictions = predictions[sorted_indices]

# 7. 保存预测结果到txt文件
output_filename = "10205102417-predictions2.txt"  # 根据实际学号修改
with open(output_filename, "w") as file:
    for filename, prediction in zip(sorted_filenames, sorted_predictions):
        file.write(str(filename) + "\t" + str(prediction) + "\n")

print("Predictions saved to:", output_filename)
```

四、 实验结果及分析

1. KNN

分别选取不同取值的 **k** 参数（临近点个数），在测试集上测得准确率。

k取值为[1,2,3,4,5,6,7,8,9,10,12,15,20,30,50,75,100]，对应准确率如下图所示



结论

可以看到，KNN算法对于手写数字分类准确率还是相当高的，均在90%以上。随着**k**值的增大，准确率先下降，后上升至最高水平。k取值在6~30之间，保持较高的水准。超过30后，准确率缓慢下降。

对于本次实验所用的数据集，k取值在7~25之间效果最佳

分析

由于手写数字特征较为明显，且各类别直接差异较大，因此分类较为容易。

当k取值较小时，增大k值，会引入噪音点，因此造成准确率的下降；随着k值继续增大，可以参考周围更多的点，减小了噪音点的影响，因此准确率回升，并且在区间内保持较高的水准；k值继续增大到一定阈值后，由于测试集数量的限制，引入更多的周围临近点后，然而会产生负影响，因此准确率下降。

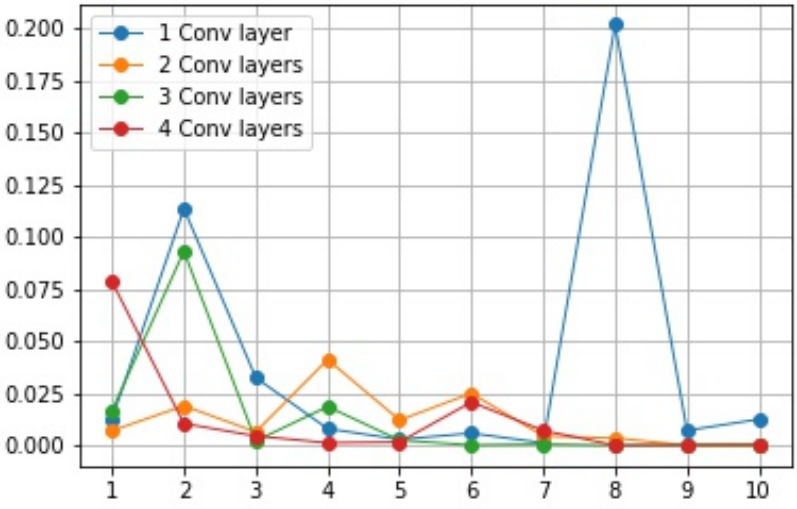
实验结果与理论分析相符合。

2. CNN

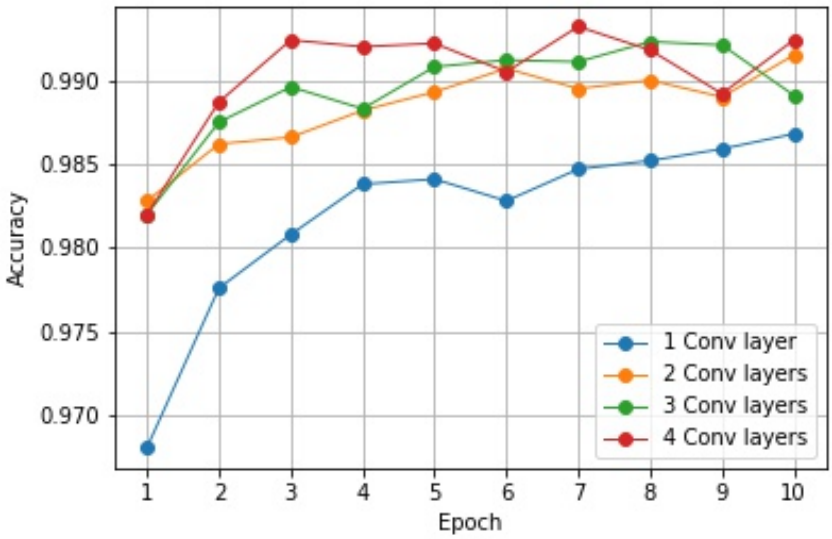
对于构建的单层卷积、双层卷积、三层卷积、四层卷积CNN模型进行训练和测试，训练10个EPOCH，得

到结果如下：

- **Loss**



- **Accuracy**



结论

可以看到，CNN整体的准确率由于KNN。

CNN模型训练一个EPOCH后基本就可以达到收敛，损失均在0.1以下，准确率已经超过KNN。

随着训练EPOCH次数增加，Loss整体呈下降趋势，Accuracy整体上升，略有小波动。

随着卷积层数量的增加，Loss并没有明显规律，但Accuracy呈上升趋势，但增量不大。

对于本次实验，采取两层卷积+两层池化的网络结构能较好的兼顾训练时间和准确率

分析

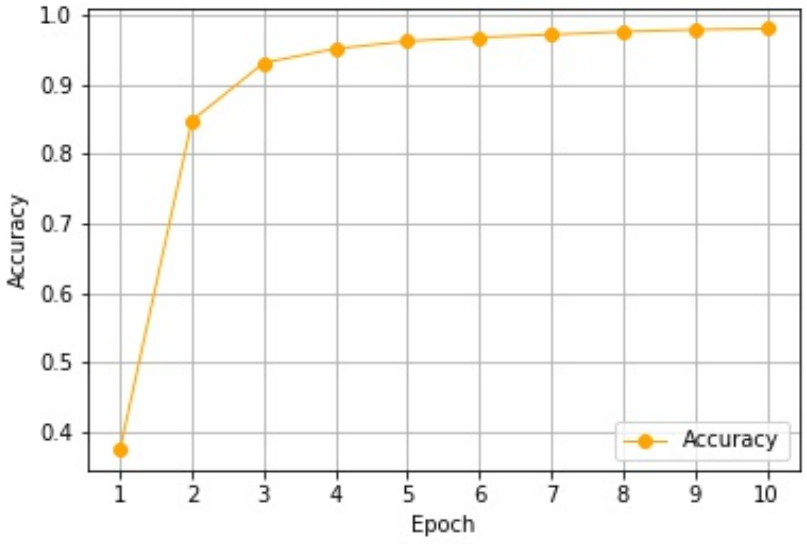
增加卷积层的数量，能够加强特征的抽取。

但由于基准准确率（baseline）已经足够高，增加卷积层数量对于Accuracy的提升收益不大。

3. LSTM

对于构建的LSTM模型进行训练和测试，训练10个EPOCH，得到结果如下：

- **Accuracy**



结论

可以看到，LSTM模型收敛较慢，大约3~4个EPOCH后，才能达到较高的准确率和较小的损失

分析

LSTM为抽取时间序列特征所设计的长短记忆单元，在处理一般图片分类任务时，起不到有效作用

4.MLP 和 随机森林

因为时间、精力原因，MLP和随机森林的训练数据只用了老师发的数据集，也**没有**对**训练迭代次数**、**模型参数**对进一步研究

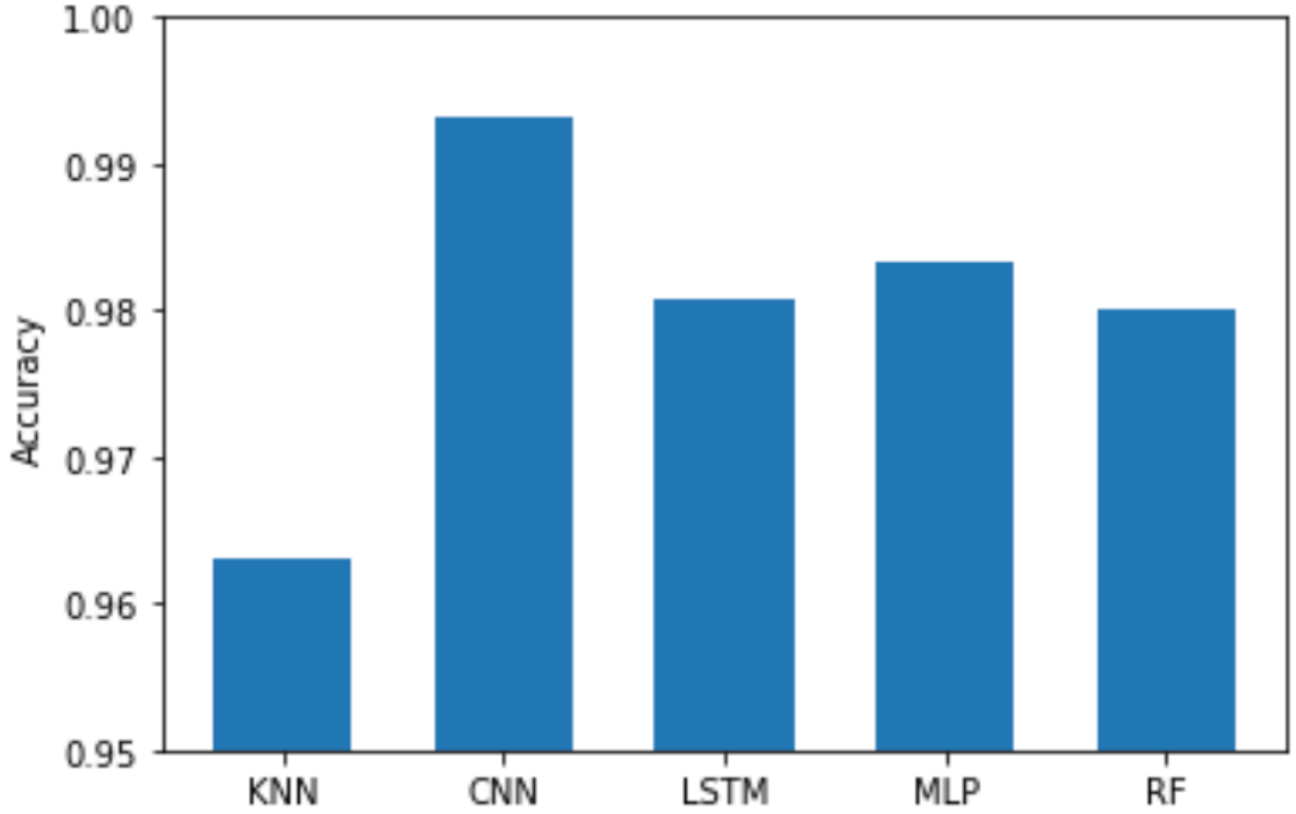
MLP，第一个隐藏层包含128个神经元，第二个隐藏层包含64个神经元，最大迭代次数为500时，准确率在**0.9833333333333333**

随机森林，在随机森林中决策树的数量为100时，准确率为**0.98**

5. 对比

将KNN、CNN、LSTM、MLP 和 随机森林5种模型的最高准确率 ([0.96300, 0.99320, 0.980794,0.983333,0.98]) 进行对比，得到如下可

视化结果



结论

可以看到，CNN效果最优，LSTM、MLP、RF其次，KNN效果相对较差，但这5个模型的准确率都相当高，均达到了95%以上

分析

KNN算法进行分类单纯取决于被分类点和周围点的距离，二CNN和LSTM可以抽取、学习到不同类别数字图片的特征信息，并利用特征信息进行分类。因此，神经网络模型CNN和LSTM的准确率要高于KNN。

LSTM为抽取时间序列特征所设计的长短记忆单元，在处理一般图片分类任务时，起不到有效作用。

LSTM效果不如单纯提取图片特征信息的CNN。

五、 问题讨论

第一次真正意义上动手实现神经网络，上手确实有一些困难。因为之前接触过pytorch，tensorflow一点也不懂，就选用了pytorch作为工具。读取数据，转化为pytorch张量，划分数据集，定义模型，训练模型，得出预测结果。一步一步做下来，确实学习到了很多，把之前人工智能课上学过的知识真正动手实践了，最主要的是初步熟悉了pytorch这个很好用的工具。

六、 结论

1. 本次实验中，采用了KNN、CNN、LSTM、MLP 和 随机森林5种模型，最高准确率 ([0.96300, 0.99320, 0.980794,0.983333,0.98])

模型效果：CNN效果最优，LSTM、MLP 和 随机森林其次，KNN效果相对较差。

2. KNN模型中，k值取值范围在7~25之间，准确率处于最高水平

3. CNN模型中，两层卷积加两层池化的网络结构（**卷积****+ReLu+池化** **—> 卷积****+ReLu+池化** **—> 全连接**），能较好的兼顾模型训练时间和准确率